Algebra of the secure channels

Author: Juan Manuel Dato Ruiz

jumadaru@gmail.com

Abstract. Some Theorems could be false, some experts could be in a wrong. But when security falls it would finish in a crysis. That is the reason we cannot use the complexity of maths to ensure something is hidden, we need to use the easy maths to ensure that there is no way to discover anything. And that is the plot of this document, with the code in Python 3.0.

1. The perfect codding.

Getting the perfect codding requires a mathematical interest; despite of all it is fascinating how many things we can do with that skill. In fact, those mecanisms can be used to sign digitally. When we use the binary we can play with the XOR function. That function works well to get a symetric cypher where the possibility of guessing the original text will be close to 0%, but the requirement to get this is using an infinitum length key.

At this point, we can study other mecanisms in different numerical bases where we could find the same property with XOR, but granting smaller keys. To get that effect, we have to study every simetries of a text using the idea of latin square.

A latin square is a square matrix of numbers positioned where there is no repetition of them in each row and column and without using more numbers than the number of rows. An example of latin square could be the result of add the values of row and column coordinates in Z_N, like in Figure 1.1.

0	1	2	3
1	2	3	0
2	3	0	1
3	0	1	2

Figure 1.1 An example of latin square of 4 elements

As you can appreciate the number of rows in Figure 1.1 is four, so those numbers will work in Z_4, and no number has a repetition in each row and column. In fact, examining the figure, we can say that for each row we can reach four interpretations of its value: in row 0 the key 0 codes the row like 0, and in row 1 the key 2 will code the value like 3.

So this mecanism is a way to code data simetrically. However, if we use a system like this and someone discovers what is the square we use then guessing the original message would be easy: He could find the most used character (study of entropy od Shanon) for supposing a meaning; and if someone guess a character, with the same square, he could easily guess the rest of the message. So using only this latin square is useless, for that reason I will call this latin square (the latin square that is the sum of the coordinates) the latin square 0. From this one we will code the rest.

For that reason we need to study this problem deeper: we need to discover all the latin squares possible in a numerical base. In this way the choice of the latin square would be in our own key, or in text.

If we start with the latin square formed by binary numbers, we will discover very easily there are only two latin squares: XOR and its negated. So, the choice of the latin square needs as much information as we need to code. In conclusion we will study base 3.

To study the number of latin squares in base 3, we must watch no repeating two codifications of the same latin square, elsewhere we will assignate two different keys to the same data. For avoiding those collisions we will use a policy: let's see how to relationate two squares whose coordinates represent a permutation (the values of the squares substitutes the coordinates like a permutation).

As we can see, that problem initially is a little complex, then we will reduce the weight: we will look for the latin square that needs the latin square 0 to construct the pair cited before. And it is from this point where we begin our study of simetries. So, from now to the rest of the document we will come back on the definition of the pair more generally.

Lemma 1.1 If in a latin square of 3 rows is permuted a row, then the result will be a pair.

That is: we have a latin square, we permute a row, and now we have two latin squares with the property that they are a pair. Its demonstration is so easy so probing for each three cases. So we can use the next **notation**: the operation **0** will mean that we permutate row 1 with 2, so the operation **1** and **2** we can guess their meaning. So we found three different pairs from the 0 latin square.

To study deeper we need more operations that will transform or not the square keeping the property of being a latin square. So that another operation is the transpost: if we change rows with columns the latin square will keep being latin square. And more: if we permutate two of the values of a latin square keeping only one x, we can note those three operations like d_x (d_1, d_2, d_0) and they will generate other latin square different.

0	1	2	d_1	2	1	0
1	2	0		1	0	2
2	0	1		0	2	1
0	1	2	d_0	0	2	1
1	2	0		2	1	0
2	0	1		1	0	2
0	1	2	t	0	2	1
2	0	1		1	0	2
1	2	0		2	1	0

Some examples of our new notation

At this point we will start showing a set of theorems:

$d_0 0 = d_1 1 = d_2 2$
$d_1 0 = d_2 1 = d_0 2$
$d_2 0 = d_0 1 = d_1 2$

Theorem 1.1

How to demonstrate it is too easy: probing each case. In fact, there is a "not in a languaje" reason in all operations, but if you do not see the reason you can probe it.

$d_0 d_1 = 02$
$d_0 d_2 = 01$
$d_1 d_0 = 12$
$d_1 d_2 = 10$
$d_2 d_0 = 21$
$d_2 d_1 = 20$

Theorem 1.2

At last we will see other group of theorems that will help us doing group operations:

$d_1 2t = 2$
$d_1 0t = 0$
$d_1 1t = 1$

Theorem 1.3

$d_0 = 0t0$
$d_1 = 1t0$
$d_2 = 2t0$

Theorem 1.4

Those theorems should be used with the next lemmas to generate the new algebra of the second part of the document.

Lemma 1.2 The rules (symbols of our notation) started from the zero (0 latin square) that are like 0t, 1t, or 2t construct three different pairs (latin squares pairs) to the cited in lemma 1.1 and theirselves.

Lemma 1.3 The rules from the zero that are like $d_0 0$, $d_1 0$ or $d_2 0$ construct three different pairs to the cited in lemma 1.2 and 1.1 and theirselves.

Lemma 1.4 We cannot construct more pairs than in lemmas 1.1, 1.2 or 1.3.

To demonstrate lemma 1.2 or 1.3 is easy probing each case. To demonstrate lemma 1.4 we will need all of the three theorems remembering that transport of zero is zero, or that double trasnport is like not doing anything. In this way, when we get any string, at last it will be reduced easily to the 9 combinations showed.

So**,** if we get the 9 combinations of the pair of the zero and we add the combinations that cannot generate a pair, that is adding X mod 3 for each value in the square we will find all the 12 latin squares. The X in my notations wil be called modificator, and it is always studied aparted.

2. Algebra generated

With the last study we will get the basics for creating two new operators that will help us to create hash functions with very hopefully simetries properties. Therefore, we will proceed testing how the operators should be definited and the kind of properties we are expecting.

For the understanding of the values of transformation of those operators we must first understand some simetries in a analisys way: the next lemma 2.1 will be our first conclussion of the first part of this document about the 12 latin pairs.

Lemma 2.1 Every latin square of size 3 can be generated with an expresion of lambda calculus: λx λy (A·x + B·y + C) mod 3, where C is a value in 0, 1 or 2, but A and B only can get the values 1 or 2.

From the last lemma, we have the 12 latin squares trapped in a modular expression. And we can order the formulas in that way:

	+0	+1	+2
X + Y	0 1 2 1 2 0 2 0 1	1 2 0 2 0 1 0 1 2	2 0 1 0 1 2 1 2 0
X + 2·Y	0 2 1 1 0 2 2 1 0	1 0 2 2 1 0 0 2 1	2 1 0 0 2 1 1 0 2
2·X + Y	0 1 2 2 0 1 1 0 2	1 2 0 0 1 2 2 0 1	2 0 1 1 2 0 0 1 2
2·X + 2·Y	0 2 1 2 1 0 1 0 2	1 0 2 0 2 1 2 1 0	2 1 0 1 0 2 0 2 1

Starting from these combinations, our operators won't work over the latin squares itselves, elsewhere with the pairs. So we can define what a latin square pair is.

Definition 2.1 A pair (latin square pair) are two latin square, called numerator and denominator, where those are transformed from zero and its pair.

So if the transformation of the zero to its pair finish in a configuration, we will find that not all numerators could be the expression of the pair of the denominator.

Therefore, as example we can see the pair of latin square "X + Y" and "X + 2·Y". They generates a pair because for every coordinate X and Y there will be an only one pair of values, and from every pair a coordinates X and Y.

In ("X + Y" / "X + 2·Y")

(X=0,Y=0) corresponds with (0, 0),

(X=1, Y=2) corresponds with (0, 2),

(X=2, Y=1) corresponds with (0, 1)

Examples of correspondences in a pair

Choiced notation.

At this point we have to find the pairs in an easy notation. We can see that the origin of the pair is the zero, and there are only 3 X 2 posibilities in the numerator, that is when numerator is "X + 2·Y" or "2·X + Y", so now we will choice the numbers to codificate the squares:

In the next scheme we can see 4 kind of latin square (with their modificators), and other combinations changing the coeficients in the lambda formula in Z_3.

		+	-
0	0 1 2 1 2 0 2 0 1	1 2 0 2 0 1 0 1 2	2 0 1 0 1 2 1 2 0
1	0 2 1 1 0 2 2 1 0	1 0 2 2 1 0 0 2 1	2 1 0 0 2 1 1 0 2
2	0 1 2 2 0 1 1 0 2	1 2 0 0 1 2 2 0 1	2 0 1 1 2 0 0 1 2
3	0 2 1 2 1 0 1 0 2	1 0 2 0 2 1 2 1 0	2 1 0 1 0 2 0 2 1
4	0 1 2 0 1 2 0 1 2	1 2 0 1 2 0 1 2 0	2 0 1 2 0 1 2 0 1
5	0 0 0 1 1 1 2 2 2	1 1 1 2 2 2 0 0 0	2 2 2 1 1 1 0 0 0
6	0 2 1 0 2 1 0 2 1	1 0 2 1 0 2 1 0 2	2 1 0 2 1 0 2 1 0
7	0 0 0 2 2 2 1 1 1	1 1 1 2 2 2 0 0 0	2 2 2 1 1 1 0 0 0

With that notation we can enumerate the pairs in bold in the next table with positive numbers and another kind of square-pair that will be called dispair in negative numbers:

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1/0	2/0	1/1	2/1	1/2	2/2	1/3	2/3
-1	**-2**	**-3**	**-4**	**-5**	**-6**	**-7**	**-8**
5/4	4/5	5/6	6/5	7/4	4/7	7/6	6/7

Table 2.1

The reason I define the dispairs is because they will appear to take group opperations. That is, two pairs generates a dispair, and a dispair with a pair generates a pair. We need both of them to get a group of Galois.

For interpreting the generation of a latin pair from one of the numbers in the table of above, we need to execute the next procedure:

1. The cypher is transformed in a pair using table 2.1.
2. The numerator is transformed by the permutation choiced in table 2.2.

Why we must transform the numerator? Because numerator is transformed with the denominator from zero, and notation comes from the zero pair.

1	2	3
(0 1) (2 3)	(1 3) (2 0)	(1 2)

Table 2.2

In example, cypher 5 (the fifth latin square pair) corresponds with pair 1/2 in table 1.2, and using table 2.2 we get the permutation (1 3) (2 0), so 1 is changed by 3. For that reason, we have to show latin squares 3 and 2 in this order.

In other hand, we have the posibility of combine the three modificators in the numerator and on the denominator independientelly using this notation $^{+}?_{-}$; so, if we choice the modificator (12) in the cypher 3 we will write $^{+}3_{-}$, and we will understand $1^{+}/1^{-}$.

In conclussion we can see 72 pairs of latin squares in total (and other 72 dispairs, whose meaning will appear later).

That notation brings us to the definition of the first operator:

1	2	3	4	5	6	7	8
2	1	4	3	6	5	7	8
3	6	1	8	7	2	5	4
4	5	2	7	8	1	6	3
5	4	7	2	1	8	3	6
6	3	8	1	2	7	4	5
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

Table 2.3: Inner product ·

This operator will be represent by the inner product ·, that is, it will be used to opperate latin pairs for preparing to give them a specifical use. Our inner opperator has the 1 like zero and it is not conmutative, but it is asociative and the inverse exists. The modificators added only can be studied with the value (X=0, Y=0). After studying the second opperator we will be prepared to see an example.

Our second opperator could be called extern product *, it is a simple permutation of values; to convert pairs in dispairs and viceversa:

*** = (1 -6 6 -7 7 -4 4 -1) (2 -8 5 -3 8 -2 3 -5)**

We will be able of opperating pairs and dispairs with those opperators in the easiest way and without weird techniques. That is, the complicated way to get here is because in this way we will get the best result of opperations.

More specifically, if we see a latin pair like an application that transforms a pair of numbers, we will can say combination of two of those applications generates other application. The composition of both applications was called by us like extern product *. Moreover, we can write: f(g(x)) = f*g(x) knowing that f and g represent pairs or dispairs.

For using those opperators we have to multiply from left to right, applying the permutation of * to the element on the left. In the next step, when two values would be connected without the extern product, we will use the inner product renembering the rule “minus by minus is positive”.

Example:

Lets start with the latin pairs 2, $^{+}3_{-}$ y 7_{+}, those are respectivally:

0	1	2	0	1	2	1	2	0	2	1	0	0	1	2	1	0	2
2	0	1	1	2	0	2	0	1	0	2	1	2	0	1	0	2	1
1	2	0	2	0	1	0	1	2	1	0	2	1	2	0	2	1	0

When we apply (from the most right, because we opperate from the left) in that example the coordinate (X, Y) = (2, 1) on 7_{+}, we get (2, 1). Now we apply (2, 1) on $^{+}3_{-}$ and results (1, 0), then we take (1, 0) on 2 and the result will be (2, 1) casually. So: $2 * {}^{+}3_{-} * 7_{+}$ (2,1) = (2,1)

There are some experts who have the opinion that the final result must be always a transformation, but in this case it will be not of my consideration; that is because the simetry is perfect and the probability of merging something very much and get it like in the beggining is too low. And it will be lower in the future...

Now I will show you how I opperate step by step to get the final pair, using the definitions of the opperators:

$$2 * 3 * 7 = -8 \cdot 3 * 7 = -6 * 7 = 6 \cdot 7 = 4$$

To know what are the modificators we have to apply the trace with the value (0,0):

(00) -> (01) -> (21) -> (20)

Solution: $^{-}4$

						2	1	0	0	2	1						
						1	0	2	1	0	2						
						0	2	1	2	1	0						

We can see that the modificator cannot be defined with those tables, but in the implementation the code is possible because those opperators are asociative.

Getting a group opperation, we will get the possibility to define and to calculate the inverse; that is so, the formula is:

$$\mathbf{A^{-1} = - ((A^*) (-4\ -6))}$$

That means that we apply * to the value A, and then we permutate -4 by -6 if those appear, and finallly we change the sign.

Unfortunatelly, modificators are victims of complications in the inverse opperation, so we have to apply table 2.4. In that way we can corroborate that inverse has the property**: $\mathbf{A * A^{-1} = -1}$**, because the application that never changes the value of (X,Y) is the dispair -1. And that is the meaning of a dispair: those applications that do not codificate at all anything, for that reason they are represented by negative form, to be very careful.

	00	**01**	**02**	**10**	**11**	**12**	**20**	**21**	**22**
1	00	11	22	12	20	01	21	02	10
2	00	11	22	21	02	10	12	20	01
3	00	12	21	11	20	02	22	01	10
4	00	12	21	22	01	10	11	20	02
5	00	21	12	22	10	01	11	02	20
6	00	21	12	11	02	20	22	10	01
7	00	22	11	21	10	02	12	01	20
8	00	22	11	12	01	20	21	10	02

Tabla 2.4. Modificators for the inverse

Calculing the inverse represents semantically the aplication that returns us the position where the values are, for that reason its corroboration could be very easy.

Example:

If we want to calculate the inverse of $\bar{7}_+$, the formula obligates us to do the next things:

$$-((7^*)(-4\ -6)) = -(-4\ (-4\ -6)) = -(-6) = \mathbf{6}$$

If we reach in table 2.4 the modificator 21 on the row 7 that results **01**.

So the inverse of $\bar{7}_+$ is $\mathbf{6}_+$.

2 0 1 1 2 0 0 1 2	1 0 2 0 2 1 2 1 0	For each (X,Y): (X,Y) -> (A,B) on the left (A,B) -> (X,Y) on the right	0 1 2 1 2 0 2 0 1	1 2 0 0 1 2 2 0 1

3. Certification Protocol.

One of the tools we can construct with this algebra is the digital sign aproaching the hash functions, or a new way of creating hash functions. In fact, I show a code with a technique of symmetrical codification and other one assymmetrical. In this protocol I will explain how we generates the sign and how to validate it.

This protocol converts a unsecure channel in secure. That is better than DH or RSA or elliptic curves, because those techniques are expecting for the possibility of appearing a formula that could finish the secutiry. Moreover, if a hardware is hacking the communication, it could be faster than machines used by users, and it would try possibilities of little formulas that could get him a partial solution that are only known by a few.

With this algebra those problems are simply impossible: without knowing the square we cannot know what was the key and how getting collisions (except trying). That is the fact so strongly, that we can codificate a known text by a symmetrical unknown key, and with the coded text we cannot guess what was the key. The only one way for hacking is notting each correspondence to make obsolote the key by the use. That process will be too slow if the sign is of a great length.

So if we want to understand the next protocol, we will need to develope the functions of symmetrical codification of a text and generation of a sign deppending of a date and a key before. One great idea more is to generate an asymmetrical key to generate a symmetrical key too much longer with a hash function to transform the “easy of renember” key of the user.

We can use too a function that merges the bytes of a text. That technique will be used to validate both Alice and Bob use the same key.

Initially we could think that protocol is too large, but it does not use too many steps; that is, I though to divide every step to the code could be more interesting than using abstract steps. That is the way I use to say that there is a probed code. Moreover, the functions used in the protocol can be shared with the code in python 3.0 I uploaded. The list of those functions is the fourth point.

For that reason I used different types of data: integer, string, text, coded text... In the majority of protocols the type of the data is irrelevant, so this protocol is in the lowest level possible.

Actually, the code is in Spanish (name of the functions, variables...) but if it is of your interest (not mine, this is for free) I could translate the names of the functions and variables.

Protocol

1. Alice has two secret numbers P1 y P2, integers.
2. Bob has two secret keys C1 y C2, strings.
3. Bob generates his sign for that date:
    a. C1'=[orden(X) for X in C1]
    b. C2'=[orden(X) for X in C2]
    c. F1 = FirmaTemporal(C1',date)
    d. F2 = FirmaTemporal(C2',date)
4. Alice generates some documents D1, D2, D3 string.
5. Alice codes documents and she merges them under the criteria P1.
    a. D1'=codifica(D1,9)
    b. D2'=codifica(D2,9)
    c. D3'=codifica(D3,9)
    d. DD=mezcla(D1'+D2'+D3',P1)
6. Alice sends the merge to Bob, and Bob applies the inverse of the second key; for returning to Alice the result.
    a. iF2 = invierte(F2)
    b. iD = aplica(iF2, DD)
7. Alice unmerge the coded text and she rescues the interested part (D1') now coded.
    a. iD1' = mezcla(iD, -P1)[:len(D1')]
8. Alice merges the result with the true document D1' and other for span D4'. And she send it.
    a. D4' = codifica(D4,9)
    b. DD2=mezcla(D1'+iD1'+D4',P2)
9. Bob applies the secret key C2 (F2) to DD2 (the sended).
    a. Contract = aplica (DD2,F2)
10. When Alice receives the Contract, she have to find D1' unmerging in its position with the digital sign of Bob.
    a. D1'+publicsignB = mezcla(DD2,-P2)[:...]
11. Bob shows the application of Contract with C1(F1)

    In this way the public sign of Bob will depend of the date and the Contract, and that will be the same that multiplica (F1, F2).

Alice could see that aplica(publicsignB,D1)==result sended by B

So the contract is signed, and it is signed by Bob in a date without any kind of doubt.

4. Using the code

The presented code is done for Python 3.0. This is the list of functions:

- asterisco (x)
  Extern product of a pair or dispair. x could be negative or positive.
- multiplica (N1, N2)
  Inner product of pairs or dispairs without modificators.
- modInversa(N, mod)
  Modificator of the inverse of N with his modificator.
- inversa(N,mod)
  Inverse of a pair.
- genCuadro(N,mod)
  Generates the lambda function that generates a latin square.
- muestra (generador)
  Shows a latin square from a generator: muestra(genCuadro(1,2))
- decodPar(N)
  Decodes an integer that represent a pair or dispair in a cypher and the modificator. The negative are dispairs.
- codPar(N,mod)
  Codes the integer of the pair/dispair.
- genPar(N,mod)
  Generates the lambda functions that generates pairs or dispairs.
- muestraPar(generador)
  Shows two squares from a generator:
  muestraPar(genPar(*decodPar(-1)))
- aplica(A,B,modA=0,modB=0)
  Multiply two pairs or dispairs with modificators.
- mezcla(L,P=0)
  Merge the array L under the criteria P. If P is negative, then the disorder will disapear.
- cambiaBaseModoGuion(source, baseDestiny)
  Transform symmetrically the list of integers source in a list with elements in base baseDestiny.
- retornaBaseModoGuion(destiny, baseDestiny)
  Retorns the original list transformed with the baseDestiny.
- montecarloSistema1011(N,cifras=8)
  Sistem of Montecarlo modified for avoiding the 0 substraying that value in the process of opperation.
- orden(caracter)
  Converts the alphanumeric in numeric, to make proofs.
- caracter(orden)
  Converts the numeric in alphanumeric, to make proofs.

- cifradoSimetricoPosicional (text, key)
  Secure prototype of symmetrical cyphering with two strings.
- descifradoSimetricoPosicional (textunreadable, key)
  Secure prototype of symmetrical uncyphering with two strings.
- firmaTemporal (key, date, cifras=20)
  Asimmetrical Cyphering of a key from a date written in a string.
- aplicaFirma(sign,codedText)
  Aplyies a sign to a code that comes from a text.
- multiplicaFirma(sign1, sign2)
  The result of aplying a sign on other.
- invierteFirma(firma)
  The sign inverse.
- comprobacion_inversas()
  Proof with all the cases that the inverse is all right.
- comprobacion_neutro()
  Proof with all the cases that the opperation with zero is all right.
- comprobacion_asociatividad()
  Proof very large with all the cases that the asociative operation is all right.